Rule-Based Runtime Verification

Howard Barringer
Allen Goldberg
Klaus Havelund
Koushik Sen

Overview

- Run-time Monitoring
- About EAGLE
- Enhanced Formal Testing
- Summary

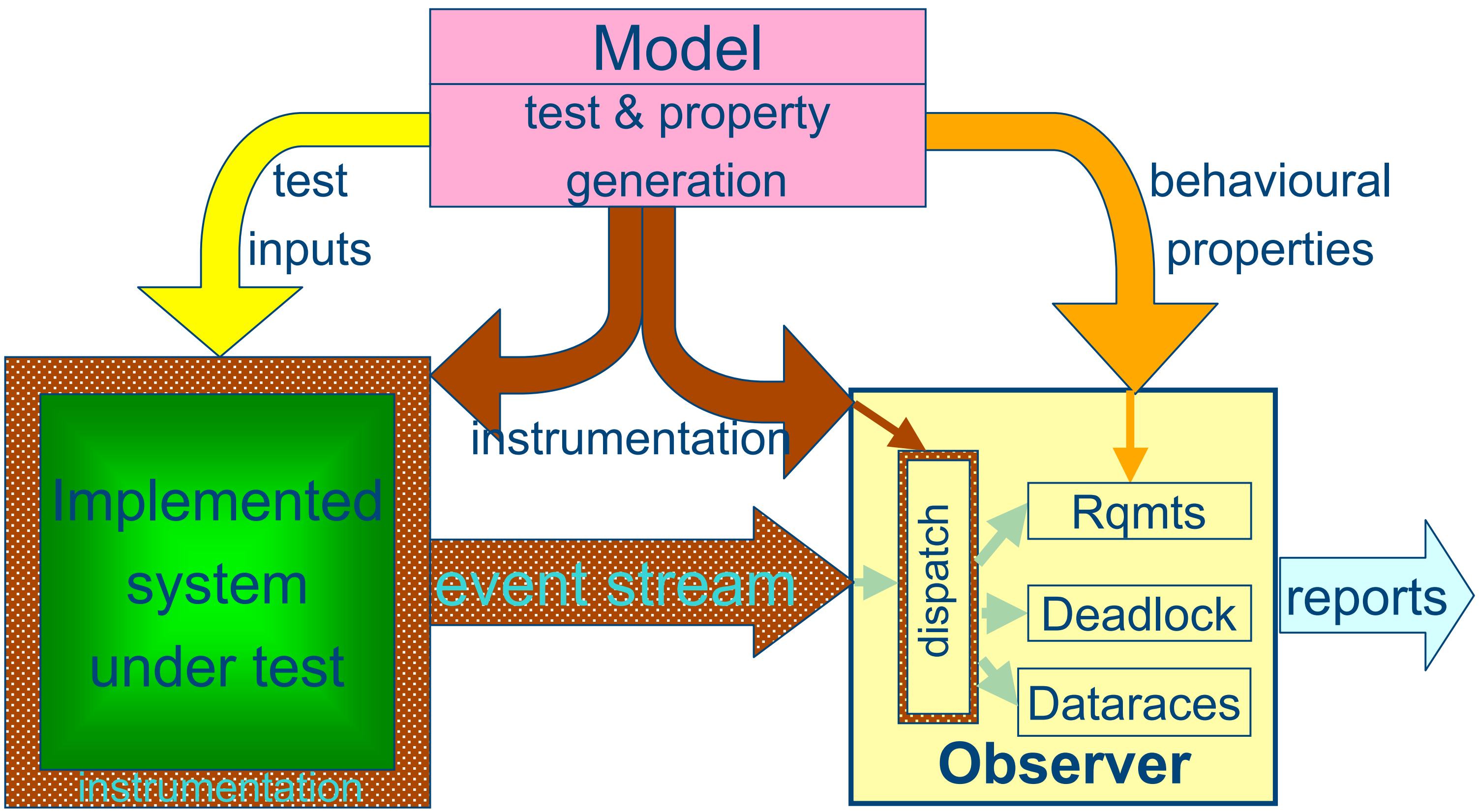
Motivation

- Model checking and Theorem Proving are rigorous
  - Not scalable
  - Complex
- Testing is scalable and widely used
  - Ad hoc
  - Lack of coverage
- Combine Formal Methods and Testing ?
  - Gain the benefits of both the approaches.
  - Avoid the pitfalls of ad hoc testing.
  - Avoid the complexity of theorem proving and model checking.

Run-time Verification

- Merge testing and temporal logic specification
  - Specify safety properties in some temporal logic.
  - Instrument program to generate events.
  - Monitor safety properties against a trace of event emitted by the running program.
- Pros: Scalable
- Cons: Lack of Coverage

A Model-Based Verification Architecture



Our work on Rqmts Monitoring

- Future time propositional:
  - Backwards dynamic programming algorithm
  - Forward rewriting algorithm (in Maude)
  - “Buchi” automata generation (Giannakopoulou)
  - BTT automata generation
- Past time propositional:
  - Forwards dynamic programming algorithm

Other Work on Rqmts Monitoring

- MaC Tool (UPenn) – uses past-time interval logic
- Temporal Rover – commercial tool
- Statistics Collection by Finkbeiner et al.
- Debugging Distributed Autonomous Systems by Simmons et al. (CMU)
- …

So many logics …

- What is the most basic, yet, general specification language suitable for monitoring?

EAGLE is our answer.

Based on recursive rules over next, previous and concatenation “temporal” connectives.

Can encode future time temporal logic, past-time logic, ERE, μ-calculus, real-time, data-binding, statistics….

Introducing EAGLE

- Rule-based finite trace monitoring logic
- User defines
  - a set of temporal rules
  - a set of monitoring formulas
- Monitors evaluated over a given input trace, on a state by state basis
- Evaluation proceeds by checking facts and generating obligations

Syntax

$$
\begin{aligned}
S &::= \textbf{dec}\ D\ \textbf{obs}\ O \\
D &::= R^* \\
O &::= M^* \\
R &::= \{\textbf{max} \mid \textbf{min}\}\ N(T_1\ x_1, \ldots, T_n\ x_n) = F \\
M &::= N = F \\
T &::= \textbf{Form} \mid \textit{java primitive type} \\
F &::= \textit{java expression} \mid \textbf{True} \mid \textbf{False} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
&\quad\ \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n)
\end{aligned}
$$

Semantics

$$
\begin{array}{ll}
\sigma, i \models_D exp & \text{iff } 1 \leq i \leq |\sigma| \text{ and } evaluate(exp)(\sigma(i)) == true \\
\sigma, i \models_D \underline{\text{true}} & \\
\sigma, i \not\models_D \underline{\text{false}} & \\
\sigma, i \models_D \neg F & \text{iff } \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff } \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff } \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff } \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff } i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff } 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff } \exists j \text{ s.t. } i \leq j \leq |\sigma|+1 \text{ and } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(F_1, \ldots, F_m) & \text{iff } \begin{cases} \text{if } 1 \leq i \leq |\sigma| \text{ then:} \\ \quad \sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\ \quad \text{where } (N(T_1\ x_1, \ldots, T_m\ x_m) = F) \in D \\ \text{otherwise, if } i = 0 \text{ or } i = |\sigma|+1 \text{ then:} \\ \quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D \end{cases}
\end{array}
$$

EAGLE by example: LTL

max Always(<u>Form</u> F) = F /\ _ Always(F) .
min Eventually(<u>Form</u> F) = F \/ _ Eventually(F) .
max EventuallyP(<u>Form</u> F) = F \/ _ EventuallyP(F) .

To monitor the LTL formula _(x>0 _ _ y=3), write

mon M1 = Always(x > 0 -> EventuallyP(y=3)) .

EAGLE by example: data binding

_(x > 0 → _k. k = x /\ _ y = 3)

can be written as

mon M1 = Always(x>0 → let k= x in Eventually(y=k)).

which is rewritten using a data parameterized rule:

min R(int k) = Eventually(y=k) .
mon M2 = Always(x>0 → R(x)) .

EAGLE by example: metric LTL

Timed operators, such as: _[t1,t2]

assume events are time-stamped _ state variable clock

min TEventuallyAbs(Form F, float t1, float t2)
= clock <= t2 ∧
(F → t1 <= clock) ∧
(~ F → _ TEventuallyAbs(F, t1, t2)) .

min TEventually(Form F, float t1, float t2)
= TEventuallyAbs(F, t1+clock, t2+clock) .

EAGLE by example: statistical logics

Monitor that state property F holds with at least probability p over the given sequence

min A(<u>Form</u> F, <u>float</u> p, <u>int</u> f, <u>int</u> t) =

(_Empty() /\ ((F /\ (1 _ f/t) >= p) \/

(¬F /\ (1 _ (f+1)/t) >= p)))

\/

(¬Empty() /\ ((F → _A(F, p, f, t+1)) /\ (¬F → _A(F, p, f+1, t+1))) .

min AtLeast (<u>Form</u> F, <u>float</u> p) = A(F, p, 0, 1) .

EAGLE by example: beyond regular languages

Monitor a sequence of login and logout events – at no point should there be more logouts than logins and they should match by the end.

min Match (<u>Form</u> F1, <u>Form</u> F2) =

Empty() \/

F1 • Match(F1, F2) • F2 • Match(F1, F2)

mon M1 = Match(login, logout)

Some EAGLE facts

- EAGLE-LTL (past and future). Monitoring formula of size m has space complexity bounded by $m^2 2^m \log m$
- EAGLE with data binding has worst case dependent on length of input trace
- EAGLE without data is at least Context Free
- EAGLE logic currently implemented by rewriting as a Java application

EAGLE: Internal Calculus

Uses four functions

init: Form X Form X Form -> Form

transforms a monitor formula (1^{st} arg) for evaluation, in particular the primitive _ and _ are replaced by rules Next and Previous with history parameters introduced to past-time rules

eval: Form X State -> Form

applies the given state to the formula yielding the obligation for the future

update: Form X State X Form X Form -> Form

updates the past time components in the formula (1^{st} arg)

value: Form -> Bool

yields the value of the given formula at the end of monitoring

EAGLE: Internal Calculus – eval - I

eval«true, s» = true

eval«false, s» = false

eval«exp, s» = value of exp in state s

eval«F_1 op F_2, s» = eval«F_1, s» op eval«F_2, s»

eval«¬F, s» = ¬eval«F, s»

eval«F_1 _ F_2, s» = if ¬value«F_1» then eval«F_1, s» _ F_2

else (eval«F_1, s» _ F_2) \/ eval«F_2, s»

EAGLE: Internal Calculus – eval - II

eval«Next(F), s» = update «F, s, null, null»

Evaluation of a next time formula Next(F) yields the obligation to evaluate F in the next state. Note that any past time args are updated by application of update

eval«Previous(F, past), s» = eval«past, s»

Since past is the (possibly partial) evaluation of F from the previous state, the evaluation of a previous time formula must just re-evaluate past in the current state

The cases of eval for rule definitions are synthesised from the rules

EAGLE: Internal Calculus – eval - III

Given rule: {**max**|**min**} R(<u>Form</u> f, <u>T</u> p) = B

a call: R(F, P)

is transformed to: <u>**R**</u>(_b.H(b), P)

where H is the transformed version of B with formal formula parameters f replaced by the transformed actual formulas F, the actual data parameters P appear as argument to <u>**R**</u> and any recursive calls to R with the same actual formula arguments are replaced by the recursion variable b

E.g.

Always(Eventually(x>0))

is transformed to:

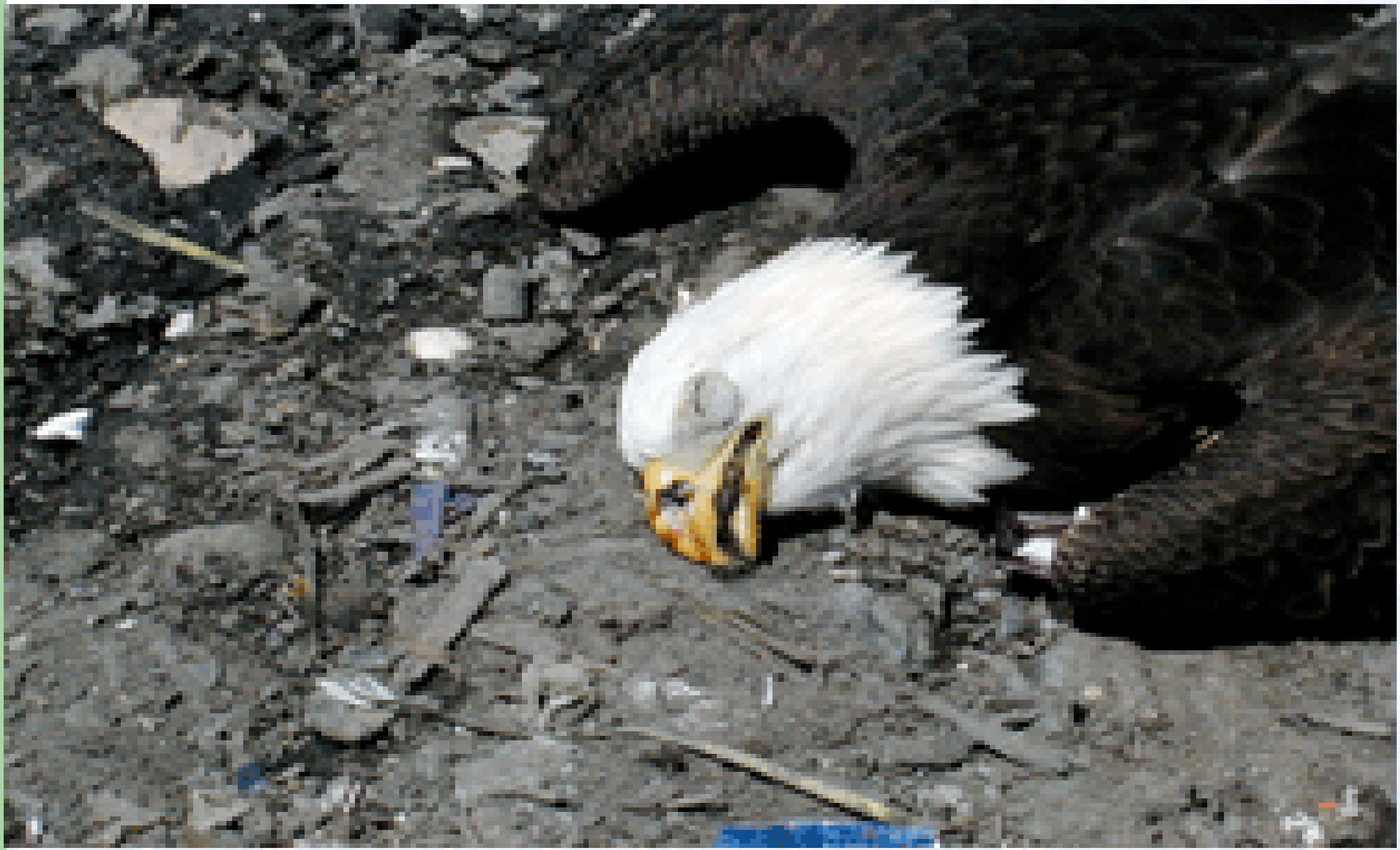
<u>**Always**</u>(_b_1. <u>**Eventually**</u>(_b_2 . (x>0) $\vee$ <u>Next</u>(b_2))) $\wedge$ <u>Next</u>(b_1))

Then the evaluation is synthesised according to:

eval«<u>**R**</u>(_b.H(b), P), s» = eval« H(_b.H(b))[p _ eval«P, s »], s»

the recursion is unfolded once, formal data parameters are substituted by the evaluated actuals, and then the whole re-evaluated.

EAGLE: Internal Calculus – eval - III



Example Execution

_(x > 0 _ _ x = 0)

Specification:

max A(Term f) = f /\ @ A(f) .

min E(Term f) = f \/ @ E(f) .

monitor M = A({x} > {0} _ E({x} == {0})).

Trace:

x=1

x=2

x=0

x=3

Trace Evaluation

_(x > 0 _ _ x = 0)

Formulas: [A(((x > 0) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ Next(A(rec)) ++ (x > 0) /\ Next(A(rec)) ++ Next(A(rec))))]

state = {x=1}

_ x = 0 °_(x > 0 _ _ x = 0)

E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ A(((x > 0) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ Next(A(rec)) ++ (x > 0) /\ Next(A(rec)) ++ Next(A(rec))))

state = {x=2}

_ x = 0 __(x > 0 _ _ x = 0)

A(((x > 0) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ Next(A(rec)) ++ (x > 0) /\ Next(A(rec)) ++ Next(A(rec)))) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec))))

state = {x=0}

_(x > 0 _ _ x = 0)

A(((x > 0) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ Next(A(rec)) ++ (x > 0) /\ Next(A(rec)) ++ Next(A(rec))))

state = {x=3}

_ x = 0 __(x > 0 _ _ x = 0)

E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ A(((x > 0) /\ E(((x == 0) ++ (x == 0) /\ Next(E(rec)) ++ Next(E(rec)))) /\ Next(A(rec)) ++ (x > 0) /\ Next(A(rec)) ++ Next(A(rec))))

Warning: Property M violated.

Correctness of EAGLE calculus

Theorem:

$$s_1,s_2,\ldots s_n,\ 1 \models_D F$$

iff

$$value(eval(\ldots eval(eval(init(F,null,\ null),\ s_1),\ s_2)\ldots,\ s_n)) = true$$

for all state sequences $s_1..s_n$ and formulas F

EAGLE: Implementation - I

- Initial implementation as a Java application
- Two phases:
  - System compiles the rule and monitor specification file to generate a set of Java classes, one for each rule and monitor
  - System then compiles the generated class files to Java bytecode and runs the monitoring engine on a given input trace

EAGLE: Implementation - II

- For efficiency, we use the propositional decision of Hsiang, where formulas are represented in Exclusive Or normal form, which is exclusive or of conjuncts.

- We use the following rewrite rules:

  F /\ F = F .

  false /\ F = false .

  true /\ F = F .

  ¬ F = true _ F .

  false _ F = F .

  F1 /\ (F2 _ F3) = (F1 /\ F2) _ (F1 /\ F3) .

  F1 \/ F2 = (F1 /\ F2) _ F1 _ F2 .

EAGLE interface

User defines these classes

e1 e2 e3 …

```
class Event {
    int x,y;
}
```

```
class Observer {
  Monitors mons;
  State state;
  eventHandler(Event e){
      state.update(e);
      mons.apply(state);
   }
}
```

```
class State {
   int x,y;
   update(Event e){
      x = e.x;  y = e.y; }
}
```

```
class Monitor {
   Formula M1, M2;
   apply(State s){
      M1.apply(s);
      M2.apply(s); }
}
```

Summary

- EAGLE is a succinct but highly expressive finite trace monitoring logic
- EAGLE can be efficiently implemented, but users must remain aware of expensive features
- Demonstrated one use by integration within a formal test environment, showing the benefit of novel combinations of formal methods and test
- EAGLE can reach parts model checking can’t
- EAGLE is almost an executable logic – can handle very limited form of action in current version

Future Work

- Optimisation of implementation – especially regarding partial evaluation
- Support user-defined surface syntax
- Associate actions with formulas – towards aspect oriented programming??
- Consider integration of EAGLE with algebraic specs
- Incorporate automated program instrumentation
- Fly EAGLE over Rainbow
- Consider Economic EAGLE – apply it to streams of economic data

EAGLE – Internal Calculus - update

update«true, s, Z, b» = true

update«false, s, Z, b» = false

update«exp, s, Z, b» = exp

update«F_1 op F_2, s, Z, b» = update«F_1, s, Z, b» op update«F_2, s, Z, b»

update«¬F, s, Z, b» = ¬update«F, s, Z, b»

update«F_1 _ F_2, s, Z, b» = update«F_1, s, Z, b» _ F_2

update«Next(F), s, Z, b» = Next(update«F, s, Z, b»)

update«Previous(F, past), s, Z, b» = Previous(update«F, s, Z, b», eval«F, s»)